Java Magazine

Coding, Java SE, JVM Internals

The Unsafe Class: Unsafe at Any Speed

Ben Evans | May 4, 2020

Just because you can break the rules, doesn't mean you should break the rules—unless you have a good reason.

Download a PDF of this article

From time to time, you may need to break the rules. In the Java platform, this is normally accomplished by using one of three primary mechanisms: reflection, class loading (including associated bytecode transformation), and `Unsafe`.

Java power users should understand all three of these techniques, even if they resort to them only when necessary. The principle "just because you *can* do something, does not mean that you *should*" applies to design choices in software as much as it does elsewhere.

Of the three, `Unsafe` is the most potentially dangerous (and therefore powerful) because it provides a way to do certain things that are otherwise impossible and that break well-established rules of the platform.

For example, `Unsafe` allows developers to

- Directly access CPU and other hardware features
- Create an object but not run its constructor
- Create a truly anonymous class without the usual verification
- Manually manage off-heap memory
- Do many other seemingly impossible things

The Java 8 `Unsafe` class, `sun.misc.Unsafe`, warns us of its very nature immediately, not only in its name, but also by the package in which it lives. The `sun.misc` package is an internal, implementation-specific location, and Java code should never touch it directly. In Java 9 and later versions, this backstage nature is made even clearer because the functionality of `Unsafe` moved to a module called `jdk.unsupported`.

Java libraries are, of course, not supposed to couple directly to these types of implementation details. To reinforce this standpoint, the attitude of Java's platform maintainers has long been that to do so is dangerous and that application developers who break the rules by linking to internal details do so at their own risk.

Despite these obvious red flags, the inconvenient truth of `Unsafe` is that it has become ubiquitous. Virtually every major framework in the Java ecosystem relies on one or more capabilities provided by `Unsafe`. It is not an exaggeration to say that the dynamism, flexibility, and

performance that modern Java developers have come to expect is largely derived from usage of `Unsafe` in one form or another.

Accessing Hardware CPU Features

Let's examine a classic use of `Unsafe` for the use of the hardware feature known as "compare and swap" or CAS. This capability is present on virtually all modern CPUs, but famously is not a part of Java's memory model.

In my first example, I'll stick to the rules and use synchronization, which forms a part of Java's memory model:

```
public final class SynchronizedCounter implements Counter {
    private int i = 0;

    @Override
    public synchronized int increment() {
        return i = i + 1;
    }

    @Override
    public synchronized int get() {
        return i;
    }
}
```

Copy code snippet

Now, let's compare that to the `AtomicCounter Unsafe` implementation, which contains significantly more boilerplate, due to having to access the `Unsafe` class reflectively:

```
public final class AtomicCounter implements Counter {
    private static final Unsafe unsafe;
    private static final long valueOffset;

    private volatile int value = 0;

    static {
        try {
            Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);
            unsafe = (Unsafe) f.get(null);
            valueOffset = unsafe.objectFieldOffset
                (AtomicCounter.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    @Override
    public int increment() {
        return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
    }

    @Override
    public int get() {
        return value;
    }
}
```

Copy code snippet

The performance difference between these two is considerable: The `Unsafe` implementation runs roughly two-to-three times faster on modern hardware.

Important reminder: As I already discussed, virtually all modern frameworks already use `Unsafe` behind the scenes. As an application

developer, you should not assume that your code will see any performance benefit whatsoever if you code against `Unsafe` directly. The framework developers have already taken advantage of `Unsafe` whenever it makes sense and do so in a safe manner. Therefore, you should use whatever your framework of choice provides.

If you trace down into the code of `AtomicCounter` and its supporting code in the JDK, you can see that this implementation is doing several things that are supposedly impossible in Java.

First of all, it computes a pointer offset (the offset where the field `value` lives relative to the start of `AtomicCounter` objects). There is no sequence of JVM bytecode instructions that can provide this; only native code that directly accesses the JVM's internal data structures can do it.

Next, it directly accesses memory via pointer offset, not by field indirection. Finally, it chooses the memory access mode (in this case, `volatile`) in which to do so—instead of having that be determined by how the variable is declared.

This isn't quite "six impossible things before breakfast," but it's pretty close!

Strictly speaking, the code breaks the rules of the Java specification by doing all this, because it is using internal capabilities that do not necessarily follow the rules in the way that user code is supposed to.

However, in practice, everyone knows that no Java developer is really willing to significantly degrade performance simply to maintain strict adherence to the Java spec. The real problem is that the `Unsafe` capability is altogether too easy to access and developers should not be encouraged to use it.

This, therefore, leads to a fundamental question: How can Java get rid of the `Unsafe` class while keeping the gains in performance and functionality that `Unsafe` has enabled and upon which the modern Java ecosystem now relies?

How Can We Get Rid of Unsafe?

Given the extent to which `Unsafe` is used (directly or indirectly) in modern Java applications, these are the only choices:

- Concede that some internals have become a de facto part of the Java API and support them as standard
- Remove access to `Unsafe` and risk breaking unknowable numbers of Java applications

In practice, this is no choice at all. Removing access without due notice and consideration would risk a de facto fork in the platform (such as that between Python 2 and Python 3, or worse).

If access to `Unsafe` were simply shut off, it would prevent almost all Java applications from upgrading, because essentially any nontrivial application uses a library that has a transitive dependency on functionality in `Unsafe`.

Related to this (and often underappreciated) is the need for the Java platform development team to be able to change internal implementation details freely, without worrying that end user Java applications (or the libraries that support them) are depending on those details.

This situation shows another essential problem at the heart of `Unsafe`. On the one hand, in recent versions, Java has evolved from a platform that has leaky encapsulation to something that aspires to much better modularity boundaries. On the other hand, `Unsafe` has become a stumbling block in the adoption of a modules system.

Solving this requires compromise. By Java 11, many of these methods have been migrated into two separate classes: `jdk.internal.misc.Unsafe` (in `java.base`) and `sun.misc.Unsafe` (in `jdk.unsupported`).

The `jdk.unsupported` module is declared like this:

```
module jdk.unsupported {
    exports sun.misc;
    exports sun.reflect;
    exports com.sun.nio.file;

    opens sun.misc;
    opens sun.reflect;
}
```

Copy code snippet

This declaration provides access for any application that explicitly depends upon the unsupported module and, crucially, it also provides unrestricted reflective access to `sun.misc`, the package containing `Unsafe`.

unrestricted reflective access to `sun.misc`, the package containing `Unsafe`.

Although this helps move `Unsafe` into a more module-friendly form, you might legitimately ask, "For how long should this compromise solution be maintained?"

To really get rid of `Unsafe`, the reflective access encapsulation hole needs to be closed, and this is probably best addressed by a change in the general reflection policy. When Java 9 was under development, the widespread usage of unrestricted reflection meant that a compromise position of defaulting to `--illegal-access=permit` was chosen.

This was intended to be only temporary, however. So, let's revisit the necessity of both of these temporary compromises.

What Has Been Done So Far?

Some methods have actually been removed from `Unsafe`, including

- `fieldOffset()`
- `monitorEnter()`
- `monitorExit()`
- `tryMonitorEnter()`

The `defineClass()` method was also moved to the `MethodHandles` class as part of Java 9.

Another major step forward is the introduction of the VarHandle API, which was added in Java 9 largely to provide safe replacements for some of the APIs in `Unsafe`.

One important goal of `VarHandle` is to include replacements for CAS functionality and access to volatile fields and arrays. To see this in action, let's look at a quick example that shows how you might approach using `VarHandle` to replace `Unsafe` in an atomic counter:

```
public class AtomicVHCounter implements Counter {
    private volatile int value = 0;
    private static final VarHandle vh;

    static {
        try {
            MethodHandles.Lookup l = MethodHandles.lookup();
            vh = l.findVarHandle(AtomicVHCounter.class, "value",
                    int.class);
        } catch (ReflectiveOperationException e) {
            throw new Error(e);
        }
    }

    @Override
    public int increment() {
        int i;
        do {
            i = (int) vh.getVolatile(this);
        } while (!vh.compareAndSet(this, i, i + 1));

        return i;
    }

    @Override
    public int get() {
        return value;
    }
}
```

Copy code snippet

This code is functionally equivalent to the previous version that uses `Unsafe`, but this code now uses only fully supported APIs. The key steps are

- Use a `Lookup` object to obtain a `VarHandle` for the appropriate field
- Cache the `VarHandle`
- Use the `VarHandle` to access the field, using volatile memory semantics

The use of `MethodHandles.Lookup` is a very important change. Unlike reflection, which relies upon `setAccessible()`to access private fields, the `Lookup` object has whatever permissions the calling context does, which includes access to the private field `value`.

The migration away from reflection and towards method and field handles means that a number of methods that were present in `Unsafe` in Java 8 have been removed from the unsupported API, including

- `compareAndSwapInt()`
- `compareAndSwapLong()`
- `compareAndSwapObject()`

The equivalents of these methods are now found on `VarHandle`, along with a number of useful accessor methods. There are also get and put methods for the primitive types and `Object`, in both normal and volatile access modes, as well as methods for building efficient adders, such as

- `getAndAddInt()`
- `getAndAddLong()`
- `getAndSetInt()`
- `getAndSetLong()`
- `getAndSetObject()`

Another key goal of `VarHandle` is to allow low-level access to the new memory order modes available in JDK 9 and later.

Overall, definite progress has been made in creating alternatives to the de facto APIs of `Unsafe`. For example, as well as `VarHandle`, the `getCallerClass()` functionality from `Unsafe` is now available in the Stack-Walking API (JEP 259). However, there is still more to do.

Hidden Classes

One other major area of progress is the work towards implementing *hidden classes*, as described in JEP 371. This JEP relates to one of the most common usages of `Unsafe`: the desire to create on-the-fly classes that cannot be used directly by other classes (but can be handled reflectively).

These classes have sometimes been referred to as *anonymous classes*, and the method in `Unsafe` is called `defineAnonymousClass()`. However, that term can be confusing to developers, because in the context of normal Java application code, it means a nested implementation of some interface that declares its static type to be the interface, like this:

```
public class Scratch {
    public void foo() {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("We had to do it this way before lambdas!");
            }
        };
    }
}
```

Copy code snippet

Many Java programmers know that classes like this implementation of `Runnable` are not really anonymous, because the compiler will generate a class named something like `Scratch$1`, which is a genuine and usable Java class. Although the class name is not available to Java source code, the class can be found using that name and accessed reflectively and then used just like any other class.

However, in this context of talking about `Unsafe`, I mean something different. That's why the term *hidden classes* should be used instead.

A hidden class is not truly anonymous either, because it has a name that is available by directly invoking `getName()` on its `Class` object. This

A hidden class is not truly anonymous either, because it has a name that is available by directly invoking `getName()` on its `Class` object. This name can also show up in several other places, including diagnostic, JVM tool interface, or Java Flight Recorder events. However, hidden classes cannot be found using a class loader or in any way that regular classes can be found, such as by using reflection (for example, via `Class.forName()`).

The intention is that hidden classes are named in a way that explicitly puts them in a different namespace than regular classes.

In the current version of the implementation of hidden classes (which is still being developed in the OpenJDK project), the naming scheme exploits the fact that in the JVM, class names typically have two forms. There's the binary name (`com.acme.Gadget`), which is returned by calling `getName()` on a `Class` object. And there's the internal form (`com/acme/Gadget`).

Hidden classes aren't named using that same pattern. Instead, a name such as `com.acme.Gadget/1234` would be returned by calling `getName()` on the `Class` object of a hidden class. This is neither a binary name nor an internal form, and any attempt to make a regular class that matches this name will fail.

One advantage of this naming scheme (and differentiating hidden classes in this way) is that they need not be subject to the usual vigorous scrutiny of the JVM's class loading mechanism. This fits with the overall design that hidden classes are intended for use by framework authors and others who need capabilities that go beyond the usual bulletproof checks imposed on general Java classes.

In the context of `Unsafe`, JEP 371 aims to deprecate the `defineAnonymousClass()` method from `Unsafe`, with the overall goal being able to remove it in a future release, once hidden classes are supported as part of the Lookup API.

This is a purely internal change. There is no suggestion that the arrival of hidden classes will change the Java programming language in any way, at least initially. However, the implementations of classes such as `LambdaMetaFactory`, `StringConcatFactory`, and `LambdaForms` will be updated to use the new APIs.

What Still Needs to Be Migrated to Supported APIs?

Let's consider which use cases need to be supported. Some important special cases are those of mocks and proxies. For those purposes, you can consider these to be objects with two special properties, specifically that they

- Can be substituted for the original class that is being mocked or proxied
- Are created without calling a constructor of the mocked or proxied class

Many modern libraries and frameworks make use of the capabilities of the low-level Objenesis project to achieve this. Objenesis uses different mechanisms (none that are publicly available) to achieve this, and one of them is `allocateInstance()` from `Unsafe`.

Therefore, to fully get rid of `Unsafe`, either this method or an equivalent capability needs to be migrated to a supported API. Java developers will still need an official way to create mocks and proxies.

This boils down to the need for an official way to instantiate an object without calling any of its class's constructors. To put it another way, space needs to be allocated for it and its instance metadata needs to be set up so the resulting object can be substituted for an instance of the original class.

For mock objects, this could be solved by creating a new `jdk.test` module that provides a special API for libraries to use when creating mocks. However, this does not solve the issue for proxies, which are used by application code at runtime, not just during testing.

This issue has also been conflated with serialization, as the current mechanism for deserialization bypasses the declared constructors of the class. A clever framework developer is thus able to piggyback mock or proxy instantiation on top of a serialization API. Creating an object that way has always been possible since the earliest days of Java.

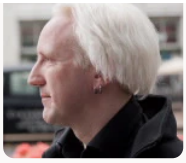
However, this approach is not sustainable for the long term. There are clear indications that at some point in the future, the serialization spec will be modified and will use constructors for deserialization, closing off a key mechanism that proxying libraries rely upon.

Conclusion

Major progress has been made in removing the dangerous parts of `Unsafe` and replacing them with standardized, supported APIs that framework and library authors can depend upon. However, although the end is in sight, the work is not done, because some functionality still relies on these unsafe methods for which there is no comparable replacement.

With luck, over the next few releases of Java, this process will be completed, and the reliance of so much of the Java ecosystem on nonstandard capabilities will end.

Ben Evans



Ben Evans

Ben Evans (@kittylyst) is a Java Champion and Senior Principal Software Engineer at Red Hat. He has written five books on programming, including *Optimizing Java* (O'Reilly) and *The Well-Grounded Java Developer* (Manning). Previously he was Lead Architect for Instrumentation at New Relic, a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.

Previous Post

The 25 greatest Java apps ever written

Alexa Weber Morales | 14 min read

Next Post

The Best of the JDK Face-Off

Sharat Chander | 7 min read